Agenda

- Characterizing the error
- Localizing the error
- Program for debugging

Reading for the Week: Chapter 13 of Matloff
The machine *does something wrong*
The original bugs were caused by moths trapped in relays

Bugs are ubiquitous in programs
Debugging is an essential and unending part of programming
Debugging is largely about differential diagnosis figuring out what has gone wrong, by eliminating other possibilities

1. Characterize the bug: figure out *exactly* what is going wrong
2. Localize the bug: find *where* the code introduces the mistake
3. Modify the code; check whether the bug has been eliminated; check that you haven’t introduced new error
Characterizing the Bug

- Make the error reproducible
  - Can we always get the error when re-running the same code and values?
  - If we start the same code in a clean copy of R, does the same thing happen?

- Bound the error
  - How much can we change the inputs and get the same error? A different error?
  - For what inputs (if any) does the bug go away?
  - How big is the error?

- Get more information
  - Add extra output (e.g., number of optimization steps, did the loop converge, final value of optimized function)
  - Much of what’s under localization below
The problem *may* be a diffused all-pervading wrongness, but often it’s a lot more localized; it helps to know where!

Tools: traceback (where did an error message come from?); print, warning, stopifnot (messages from the code as it goes)
Trying controlled inputs
Interactive debugging
Traces back through all the function calls leading to the last error. Start your attention at the first of these functions which you wrote. Often the most useful bit is somewhere in the middle (there may be many low-level functions called).
Example: Jackknife

Suppose I wrote my estimator like this:

```r
gamma.est <- function(data) {
  m <- mean(data)
  v <- var(data)
  s <- v/m
  a <- m/s
  return(list(a=a,s=s))
}
```

Now I write my jack-knifer:

```r
gamma.jackknife <- function(data) {
  n <- length(data)
  jack.estimates <- c()
  for (omitted.point in 1:n) {
    jack.estimates <- rbind(jack.estimates,gamma.est(data[-omitted.point]))
  }
  var.of.estts <- apply(jack.estimates,2,var)
  jack.var <- ((n-1)^2/n)*var.of.estts
  return(sqrt(jack.var))
}
```
What happens?

> gamma.jackknife(cats$Hwt[1:3])
Error: is.atomic(x) is not TRUE
> traceback()
5: stop(paste(ch, " is not ", if (length(r) > 1L) "all " , "TRUE", 
    sep = ";.") , Call. = FALSE)
4: stopifnot();is.atomic(x))
3: FUN(newX[, i], ...)
2: apply(jack.estimates, 2, var)
1: gamma.jackknife.2(cats$Hwt[1:3])

Tells us that the error arose from trying to apply var to each column of jack.estimates
Adding commands to the code for intermediate messages

print forces values to the screen
stick it before the problematic part to see if values look funny

```r
print(paste("x is now",x))
y <- a.tricky.function(x)
print(paste("y has become",y))
```

then add more prints upstream or downstream as needed
Add `print(str(jack.estimates))` before the `apply` and run again:

```r
> gamma.jackknife(cats$Hwt[1:3])
List of 6
$ : num 32.4  
$ : num 21.8  
$ : num 648  
$ : num 0.261  
$ : num 0.379  
$ : num 0.0111  
- attr(*, "dim")= int [1:2] 3 2
- attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:2] "a" "s"
NULL
Error: is.atomic(x) is not TRUE
```

The problem is that `gamma.est` gives a list, and so we get a weird list structure, instead of a plain array.

Re-write `gamma.est` to give a vector (as in the code provided), or wrap `unlist` around its output.
warning: print warning messages along with the call that initiated the weirdness

```r
> quadratic.solver <- function(a,b,c) {
+   determinant <- b^2 - 4*a*c
+   if (determinant < 0) {
+     warning("Equation has complex roots")
+     determinant <- as.complex(determinant)
+   }
+   return(c((-b+sqrt(determinant))/2*a, (-b-sqrt(determinant))/2*a))
+ }
> quadratic.solver(1,0,-1)
[1] 1 -1
> quadratic.solver(1,0,1)
[1] 0+1i 0-1i
Warning message:
In quadratic.solver(1, 0, 1) : Equation has complex roots
```
stopifnot: halt when results aren’t as we expect, and say why
We’ve seen this before
N.B., once you have found the bug, it’s generally good to turn lots of these off!
Test Cases and Dummy Functions

Localize error by using inputs where you know the answer
If you suspect foo is buggy, give foo a simple case where the proper output is easy for you to calculate “by hand” (i.e., not using foo)
If foo works on a bunch of cases, well and good; if not, you need to fix it (and possibly other things)
If inputs come from other functions, write functions, with the right names, to generated fixed, simple values of the right format and content
(save the real functions somewhere else)
To make sure the dummy is working, make its output as simple as you can
We want to estimate parameters by minimizing mean squared error.
Hard to say whether we’ve actually found the minimum.
Replace true MSE function with something we can minimize by hand:

```r
mse <- function(params, N = gmp$pop, Y = gmp$pcgmp) {
  return((params[1] - 6000)^2 + (params[2] - 0.13)^2)
}
```

N.B., takes all the arguments but ignores some of them.
Interactive Debugging

The browser, recover and debug functions modify how R executes other functions. Let you view and modify the environment of the target function, and step through it. You do not need to master them, though they can be very helpful. See chapter 13 of Matloff, and §§3.5–3.6 of Chambers.
After diagnosis, treatment: once the error is characterized and localized, guess at what’s wrong with the code and how to fix it. Try the fix: does it work? Have you broken something else? Try small cases first!
Parenthesis mis-matches

[[...]] vs. [...]  

== vs. =  

Identity of floating-point numbers  

Vectors vs. single values: code works for one value but not multiple ones, unexpected recycling  

Element-wise comparison of structures (use identical, all.equal)  

Silent type conversions
Common Issues: Logic

Confusing variable names
Confusing function names
Giving unnamed arguments in the wrong order
R expression does not match the math you mean (left something out, added something)
Common Issues: Scope and Global Variables

Relying on a global variable which doesn’t have the right value
(or only has the right value in one situation)
Assuming that changing a variable inside the function will change it elsewhere
Confusing variables within a function and those from where
You are going to have to debug
Debugging is frustrating and time-consuming
Writing now to make it easier to debug later is worth it, even if it takes a bit more time
A lot of the design ideas we’ve talked about already contribute to this
Comment your code

- Insist on the three comment lines for each function: purpose, inputs, outputs
- Comment the innards as well, especially anything which strikes you as tricky or clever
- If you borrowed an idea from somewhere, use the comment to remind yourself of where (and acknowledge the borrowing)

Use meaningful names

- No restrictions on name lengths, few on name content
- Avoid abbreviations, unless very well-established conventions (and put in comments explaining the convention)
Designing for Debugging

- Use top-down design and write modular, functional programs
- Respect the interfaces
- Don’t write the same code multiple times
- Use tests
Top-Down Programming

Easier to identify errors, because the job of each function is small and well-characterized
Easier to localize errors
  ● if a bottom-level function is working, the error must be somewhere up the chain
  ● if a function can integrate artificial inputs, the problem has to be either in the inputs its called with, or in a sub-function
so get the lowest-level functions right, and then work back up the chain
Respecting the interface means giving everything needed as part of the input (or context of definition) and only relying on the explicit return value.

- Makes it easier to reproduce bugs
- Makes it easier to characterize bugs by finding the bad inputs
- Global variables considered especially harmful
- Special considerations for stochastic simulations, which we’ll come to later
Unified Code

Often have to do basically similar tasks at multiple points in the program
Either write parallel code for each instance, or a single function called multiple times
Writing one function is better for debugging
- If it’s wrong, the error gets propagated everywhere
- but there is only one place that needs fixing
- and there is no chance to introduce new errors by mistakes in copying or adjustment
Tests

Helps answer “How do I know I’ve fixed this bug?”
Helps answer “How do I know I haven’t broken something that was working?”
Much of what you did to characterize and localize the bug can be turned into tests
Ordinarily, errors just lead to crashing or the like
R has an **error handling** system which allows your function to
catch, and recover from, errors in functions they call (functions:  
`try`, `tryCatch`)
Can also recover from not-really-errors (like optimizations that
don’t converge)
This system is very flexible, but rather complicated; beyond our
scope
See §3.7 of Chambers
Summary

- Debugging is largely about differential diagnosis
- When you find a bug, characterize it by making sure you can reproduce it, and figure out what inputs do and don’t give the error
- Once you know what the bug does, localize it by traceback and adding messaging from the code; by dummy input generators; and by interactive tracing
- Examine the localized error for syntax error and for logical errors; fix them, and see if that gets rid of the bug without introducing new ones
- Program for debugging: write with comments and meaningful names; write modular functions; avoid repeated code

Next time: scope